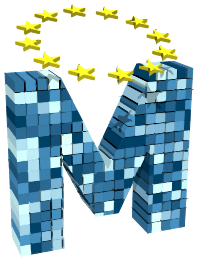


Project title: Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control
Project acronym: MOSAICrOWN
Funding scheme: H2020-ICT-2018-2
Topic: ICT-13-2018-2019
Project duration: January 2019 – December 2021

D3.4
Final Tools for the Governance Framework

Editors: Sabrina De Capitani di Vimercati (UNIMI)
Reviewers: Flora Giusto (MC)
Benjamin Weggenmann (SAP SE)

Abstract

This deliverable presents the final tools for the governance framework developed in Work Package 3 (WP3). The main goal of WP3 is the definition of a data governance framework that supports the MOSAICrOWN policy specification and manages access to data in the data market. Starting from all the relevant requirements and protection needs identified from the use cases as well as from the data market context, we have designed a policy model and language and a policy engine enforcing the access policies. This deliverable presents the main functionality of the policy engine developed to support a wider range of use cases. After a brief overview of the policy model and language, we present the architecture of the policy engine and describe the working of the modules composing it.

Type	Identifier	Dissemination	Date
Deliverable	D3.4	Public	2021.09.30

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825333.

MOSAICrOWN Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	EMC Information Systems International	EISI	Ireland
3.	Mastercard Europe	MC	Belgium
4.	SAP SE	SAP SE	Germany
5.	Università degli Studi di Bergamo	UNIBG	Italy
6.	GEIE ERCIM (Host of the W3C)	W3C	France

Versions

Version	Date	Description
0.1	2021.09.06	Initial Release
0.2	2021.09.27	Second Release
1.0	2021.09.30	Final Release

List of Contributors

This document contains contributions from different MOSAICrOWN partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Sabrina De Capitani di Vimercati (UNIMI)
Chapter 1: Policy model and language	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 2: Architecture of the policy engine	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 3: Core module	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 4: Front-End module	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 5: Back-End module	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 6: Conclusions	Sabrina De Capitani di Vimercati (UNIMI)

Contents

List of Figures

Executive Summary

This deliverable describes the main functionality implemented in the policy engine tool, providing for effective enforcement of the policy specifications. Deliverable D3.3 [DS20] has presented the first version of the engine. In line with the project ambition and requirement to provide techniques with practical applicability and compatibility with current technology, our realization of the policy engine builds on ODRL (a W3C policy specification standard). We then illustrate how MOSAICrOWN policies can be translated into ODRL specifications, addressing then different problems that raise in the actual evaluation and enforcement of policy rules. Starting with a brief introduction (Chapter 1) summarizing the main concepts at the basis of the proposed policy model and language, the core of the document is then organized in four main chapters.

Chapter 2 first illustrates the encoding of the MOSAICrOWN policy language in ODRL. The Chapter then describes the architecture of the policy engine, which is composed of three modules: Front-End, Core, and Back-End. The policy engine is written in Python and performs several tasks associated with the parsing and importing of MOSAICrOWN policies and with deriving a policy knowledge graph based on the imported policies. Access requests are analyzed against the policy knowledge graph to verify if they can be authorized.

Chapter 3 describes the Core module of the policy engine. The Core module is responsible for managing policies and for verifying whether an access request can be authorized. The chapter first provides an overview of the working of the Core module, and then describes the joint attribute visibility property used in the access control decision process. The chapter continues with a description of the main functionality implemented in the new version of the policy engine, that is, the support for hierarchies, constraints, and prohibitions.

Chapter 4 describes the Front-End module of the policy engine. The Front-End module is used to process an access request. It receives an access request written according to a specific query language as input and returns the targets of the query along with the information about the subject submitting the query, operation, and purpose. The query can be written using any query language, each having a dedicated Front-End module. In particular, we have implemented two Front-End modules: one module supports queries written in SQL, which is the universal language to access data stored in relational systems, and one module supports queries written in SPARQL, which is the query language for RDF data.

Chapter 5 illustrates the Back-End module of the policy engine. The main goal of the Back-End module is to rewrite an access request written according to a specific query language to enforce the policy constraints returned by the Core module. Like for the Front-End, the chapter describes two Back-End modules, one supporting SQL and another one supporting SPARQL. Finally, Chapter 6 concludes the deliverable.

1. Policy model and language

The goal of this chapter is to briefly summarize the elements of the policy model and language at the basis of the work on the policy engine. We first describe the main concepts of the policy model (Section 1.1), and then describe the corresponding language for expressing security requirements, focusing on how an access request is modeled and on the policy rules supported (Section 1.2).

1.1 Policy model

The basic elements of the policy models are *subject*, *object*, *operation*, *purpose*, and *condition* that are also captured by the policy language. We now illustrate these elements in more details.

1.1.1 Subjects

Subjects (data owners and consumers) are entities using the services offered by the data market and submitting or requesting access to data. The data market provider recognizes only subjects registered to the data market. Each subject *s* is assigned an identifier, denoted *id*(*s*), and a set of *properties*. To capture and reason about these properties, we assume each subject is associated with a *profile*, which defines the name and value of each property. To be as general as possible, we view profiles as semi-structured documents (e.g., implemented through XML or RDF like documents). Figure 1.1 illustrates an example of two subject profiles associated with `Alice` and `Bob`, respectively. At the abstract level, we use the classical dot notation to refer to a specific property. For instance, `Alice.email` denotes the `email` attribute in the profile of `Alice`.

Subject groups

Abstractions can be defined within the domains of subjects. Intuitively, abstractions group together subjects with common characteristics and can be referenced through a name. Groups can be nested (i.e., groups can be defined as members of other groups) and need not be disjoint (i.e., a subject can belong to more that one group). Groups and the containment relationship can be graphically represented as a directed acyclic graph, which we call subject hierarchy. The nodes of the subject hierarchy are groups and arcs represent the one-to-many containment relationship from a parent group to a child group. We assume each hierarchy to be rooted, meaning that there is one element to which all elements in the hierarchy belongs. Figure 1.2 illustrates an example of subject hierarchy, where groups distinguish different categories of subjects. `Any` is the root of the subject hierarchy. Here, for example, the `Financial` group contains groups `Advisor` and `Analyst`.

Alice

Attribute	Value
id	78934
name	Alice
dob	1990-07-22
group	Administrative
address	Ficus street, Auckland
region	Auckland
citizenship	NZ
email	alive@mymail.nz
telephone	0273874629
creditCard	678345792064
registrationDate	2021-03-01

Bob

Attribute	Value
id	78234
name	Bob
dob	2000-03-10
group	Advisor
address	Picea street, Milan
region	Lombardy
citizenship	Italian
email	bob@mytree.com
telephone	0373567914
creditCard	923735130452
registrationDate	2020-07-05

Figure 1.1: An example of two subject profiles

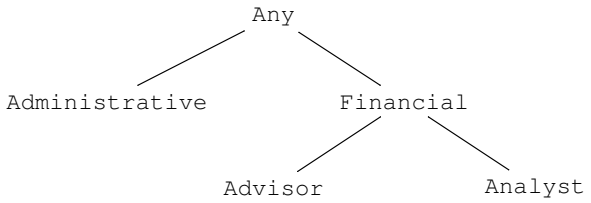


Figure 1.2: An example of subject hierarchy

1.1.2 Objects

Objects are the entities to which accesses can be requested. The Data Market distinguishes two kinds of objects: *datasets* and *metadata*.

Datasets

Datasets contain information to which access is being regulated. Our model supports both the access to a whole dataset as well as the access to a finer granularity. We consider *structured* and *semi-structured* datasets that are characterized by an unique identifier and a set of attributes modeling properties of the dataset items. Like for subjects, given a dataset d, $id(d)$ denotes the unique identifier of dataset d. Note that datasets can be stored in their original format (plaintext) or can be stored in a protected form, meaning that the data are transformed before moving them to the data market platform [DS20]. For instance, Figure 1.3(a) illustrates two datasets, `CardHolder` and `Payment` storing information about the holders of credit cards and their transactions, respectively. For simplicity, the example shows the original plaintext datasets.

Datasets can also be organized in a hierarchical structure, defining sets of datasets that can be collectively referred together with a name and that represents a *category* of datasets (e.g., financial datasets and marketing datasets). Like for subjects, categories of datasets need not be disjoint and can be nested. The definition of categories of datasets and their containment relationship introduces a hierarchy of categories, called *data category hierarchy*. Again, we assume the hierarchy to be rooted. Figure 1.4 illustrates an example of data category hierarchy with `Any` as root and two main categories, `Financial` and `Marketing`. Category `Marketing` is further specialized in `TVCampaign` and `SocialAdv` as reported in the figure.

CardHolder

CustomerID	Name	Surname	DateofBirth	Email	Marketing	CreditScore
342940	John	Smith	2000-01-31	jsmi@example.org	TRUE	650
304198	Jane	Doe	1995-06-08	jdoe@example.org	FALSE	800
257834	Mario	Bianchi	1989-10-08	mbia@example.org	FALSE	720
...	...	...	...	...	...	...

Transaction

TransactionID	CustomerID	Day	Month	Year	Amount	Merchant
68748367	342940	18	04	2020	34.50	Medical Center Ltd.
62748368	304198	03	01	2020	104.00	Shopping Mall S.A.
65748369	257834	07	03	2020	12.43	Groceries GmbH
...	...	...	...	...	...	...

(a)

META (CardHolder)
- **category** : Financial
- **status** : valid
- **creator** : BranchA
- **retention** : ten years

META (Transaction)
- **category** : Financial
- **level** : 1
- **creator** : BranchA
- **retention** : ten years

(b)

Figure 1.3: An example of two datasets (a) and corresponding metadata associated with the datasets (b)

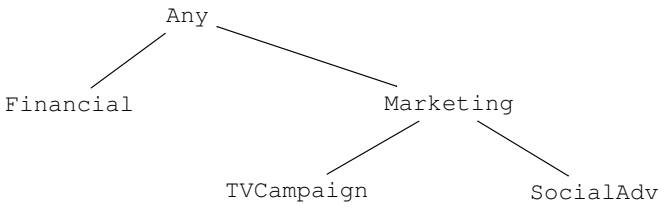


Figure 1.4: An example of data category hierarchy

Metadata

Metadata represent data about data. They can be in the form of textual or semistructured documents, and are associated with datasets. We assume that a bijective function META() makes the association between a dataset and its metadata. For instance, given dataset `CardHolder`, function META(`CardHolder`) refers to its metadata that include the information about the category of the dataset, the status of the dataset, the creator of the dataset, and the retention policy applied to the content of the dataset (see Figure 1.3(b)).

For metadata browsing as well as for the evaluation of conditions that may determine whether a given access to datasets can be allowed, it is useful to evaluate the content of metadata. We

Figure 1.5: An example of purpose hierarchy

then allow reference to finer grained content at the level of single property. Properties within a metadata document are referred by means of the classical dot notation. For instance, notation META(`CardHolder`).`category` denotes the category of dataset `CardHolder`.

1.1.3 Operations

Operations to be considered may vary depending on the specific dataset. Generally, the kinds of operations supported should include: `download` (a subject can download an object and can perform off-line analysis), `analyze` (a subject can invoke a set of pre-defined operations that perform calculations on selected objects), `browse` (a subject can browse a dataset without downloading it). Abstractions can also be defined on actions, specializing actions, or grouping them in sets. For instance, the three kinds of actions mentioned above can all be grouped in a set called `Access` and thus referred to as one.

1.1.4 Purpose

Purpose is an important concept that is mentioned in recent regulations (e.g., the General Data Protection Regulation). It represents the purpose of the processing (e.g., subjects can have the need of restricting access to their data for research purposes only). Our model captures this concept and supports the definition of abstractions over it. Different purposes can be grouped together and can be represented by a more general purpose. This is equivalent to say that purposes can be organized according to a *purpose hierarchy*. Figure 1.5 illustrates an example of purpose hierarchy rooted at `Any`.

1.1.5 Conditions

MOSAICrOWN model and language supports the definition of conditions associated with policy rules. Such conditions must be satisfied to consider the policy rules with which they are associated. In particular, we consider two kinds of conditions:

- conditions on subject profiles;
- conditions on metadata.

Such conditions evaluate properties in subject profiles and metadata associated with datasets. Their evaluation returns either true or false, depending on whether the condition is satisfied.

1.2 Policy language

We provide a brief description of the policy language for regulating access to objects stored in a data market and show the different components of the policy rules.

An access policy defines policy rules concerning access to objects in the data market. Policy rules correspond to authorizations that need to be checked when a subject submits an access request. When an access request is submitted to the data market platform, the data market first evaluates such request against the authorization policies *applicable* to it. If there exists a policy rule that matches the request, the access is permitted. If none of the policy rules apply to the access request or there are conditions in the policy rules that cannot be evaluated, the request is denied.

1.2.1 Access request

The data market provider receives requests from subjects for processing objects (we use the term "processing" in a broader sense, to indicate any operation that is performed on objects). An access request is a quadruple of the form:

$$\langle \textit{subject}, \textit{object}, \textit{operation}, \textit{purpose} \rangle$$

- *subject* is the subject that makes the request.
- *object* is the object on which *subject* wants to perform *operation*. It can be the identifier of an object or a set of attributes that characterize an object.
- *operation* is the operation that the *subject* requires to perform over the *object*. The set of available operations may vary depending on the specific object.
- *purpose* represents the reason for which object *object* is being requested. It can correspond to an element of the purpose hierarchy, meaning that it may correspond to a ground purpose or to an abstraction.

The following are examples of access requests.

- ⟨`Alice`, `download`, `CardHolder`, `Commercial`⟩: subject `Alice` requires to `download` object `CardHolder` for `Commercial` purposes.
- ⟨`Alice`, `access`, `CardHolder`.{`Name`, `Surname`, `Email`}, `Marketing`⟩: subject `Alice` requires to `access` (with `access` a generalization of both `download` and `read` operations) attributes `Name`, `Surname`, and `Email` of object `CardHolder`.

1.2.2 Policy rule

A policy includes a set of *policy rules* that regulate how the objects stored in the data market can be used by requesting subjects. A policy rule has the following form.

$$\langle (\textit{subject},\textit{condition}), (\textit{object},\textit{condition}), \textit{operation}, \textit{purpose} \rangle$$

An authorization states which *subject* can perform which *operation* on which *object* under which *condition*, and for which *purpose*. Different authorizations can apply to the same access request, and therefore the access request is granted if there is at least an authorization that allows it. We now analyze in more details the different components of the policy rule.

Subject. The *subject* component refers to a specific subject or group as well as to a set of subjects depending on conditions defined over subjects' profiles. Also, to make it possible to refer to the subject of the access request being evaluated without need of introducing variables in the language, we introduce the keyword **subject** that indicates the identifier of the subject making the request. The following are examples of *subject*.

- (`Any`,_) denotes any subject.
- (`Administrative`, **subject**.`citizenship`=EU) denotes subjects who are member of the group `Administrative` and are European citizens.

Object. The *object* component refers to a specific object or data category as well as to a set of objects depending on conditions that are evaluated on the metadata associated with the objects. Also, *object* component may refer to objects at different granularity levels: it may refer to an object as a whole as well as to any of its components. For this latter, we assume structured data characterized by a set of attributes (this can be easily extended to the consideration of semi-structured or unstructured data).

To make it possible to refer to the possible metadata associated with the object to which the request being processed refers, without need of introducing variables in the language, we introduce the **metadata** keyword, whose appearance in the object component must be substituted with the actual parameters of the access request in the evaluation at access control time. The following are example of *object*.

- (`Financial`, _) denotes all objects belonging to the `Financial` data category.
- (`Financial`, **metadata**.creator = BranchA) denotes all datasets of category `Financial` and created by BranchA
- (`CardHolder`.{`Name`, `Surname`, `Email`}, _) denotes attributes `Name`, `Surname`, and `Email` of dataset `CardHolder`.

Operation. The *operation* component corresponds to the operation to which the policy rule refers. Note that abstractions can also be defined on operations, specializing operations or grouping them in sets. An *operation* corresponds then to the identifier of a basic operation, or the identifier of an abstraction.

Purpose. The *purpose* component refers to a specific purpose or to an abstraction defined in the purpose hierarchy. Examples of purposes may be: `Any` (denoting any purpose), `Commercial`, `Statistical`, and `Marketing`.

1.2.3 Policy enforcement

Given an access request, to determine whether the request is allowed or denied the policy engine has to retrieve the *applicable* policy rules. A policy rule applies to the given access request when the subject, object, operation, and purpose of the request *are covered* by the corresponding components of the policy rule. The coverage relationship is based on the fact that policy rules defined over subject, object, operation, and purpose abstractions propagate down in the corresponding hierarchies. An access request is then permitted if there exists at least an applicable policy rule such that the possible conditions in the subject and object component of a policy rule are satisfied by the

`Alice` can `read` any object of category `Financial` for `Statistical` purposes that is created by BranchB	
subject	(`Alice`, _)
object	(`Financial`, **metadata**.`creator`=BranchB)
operation	`read`
purpose	`Statistical`

`Administrative` subjects can `read` for `Any` purposes attributes `Name`, `Surname`, and `Email` of `CardHolder`	
subject	(`Administrative`, _)
object	(`CardHolder`.{`Name`, `Surname`, `Email`}, _)
operation	`read`
purpose	`Any`

Figure 1.6: An example of policy rules

subject and object of the access request. As an example, consider the policy rules in Figure 1.6, and the access request ⟨`Alice`, `CardHolder`.{`Name`, `Surname`}, `read`, `Statistical`⟩. Given this access request, the policy engine collects all the applicable policy rules, which are all the rules in the figure (`Alice` is a member of the `Administrative` group and `CardHolder` belongs to the `Financial` category). The policy engine verifies whether the conditions in at least one policy rule are satisfied. In our example, the first policy rule in Figure 1.6 includes condition **metadata**.`creator`=BranchB that is not satisfied. In fact, the creator of `CardHolder` is BranchA (see Figure 1.3(b)). However, the second policy rule in Figure 1.6 applies to the access request and there is not any further condition to satisfy. The access request is then allowed and the system returns to `Alice` attributes `Name` and `Surname` of `CardHolder`.

2. Architecture of the policy engine

In this chapter, we describe the architecture of the policy engine tool that is responsible for parsing the MOSAICrOWN policies, written according to the policy model and language summarized in Chapter 1, and for checking whether an access request is permitted. In particular, we first recall how the MOSAICrOWN policy rules are encoded (Section 2.1), and then present the architecture of the policy engine (Section 2.2).

2.1 Encoding of MOSAICrOWN policy rules

MOSAICrOWN policies are expressed in ODRL [IV18], a W3C policy specification language that provides a flexible and interoperable information model, vocabulary, and encoding mechanism for representing statements about the usage of content and services. The choice of ODRL has been motivated by a number of considerations. ODRL has been originally designed to manage the requirements of platforms for the distribution of digital content (e.g., songs, movies). This scenario is a clear example of a wide market for digital data, where market operators let producers and consumers meet. The management of restrictions on the use of the content is of great importance for this scenario and ODRL supports an expressive representation of these restrictions. ODRL also is based on a model that is well designed and flexible. Compared to access and usage control models implemented by specific platforms and operating systems, the model is designed to be extensible, supporting its adaptation to scenarios that go beyond the representation of access privileges to multimedia files. The specification of ODRL already provides an attempt at the definition of a Privacy Profile, with an explicit consideration of the adaptability of ODRL to the MOSAICrOWN scenario. ODRL is also designed to be a high-level specification, with a variety of representations. Similar choice has been made for the MOSAICrOWN policy language, which considers alternative representations, with the one based on JSON-LD described in this chapter as the main option, but not the only one. For instance, an OWL representation is already being developed, to facilitate the application of reasoning on the policy itself.

The use of ODRL has given the opportunity to immediately adapt to MOSAICrOWN a number of components that have already been developed for ODRL, like the examples of specific vocabularies for the representation of operations and subjects. A feature of ODRL is that it is not directly related to a specific engine for the execution of the policy. This is another aspect where there is alignment with the MOSAICrOWN policy language, which aims to operate in a variety of technological domains, with data collections represented in many formats, like JSON-LD objects, tables in classical relational databases, and Data Frames supported by the Apache Spark big data frameworks. The efficiency requirements deriving from the management of large data collections mandate the design of a number of ad-hoc solutions for each domain.

The major limitation of ODRL is the relatively limited support it receives in current digital multimedia markets. We argue this is due to the fact that the specification has been recently

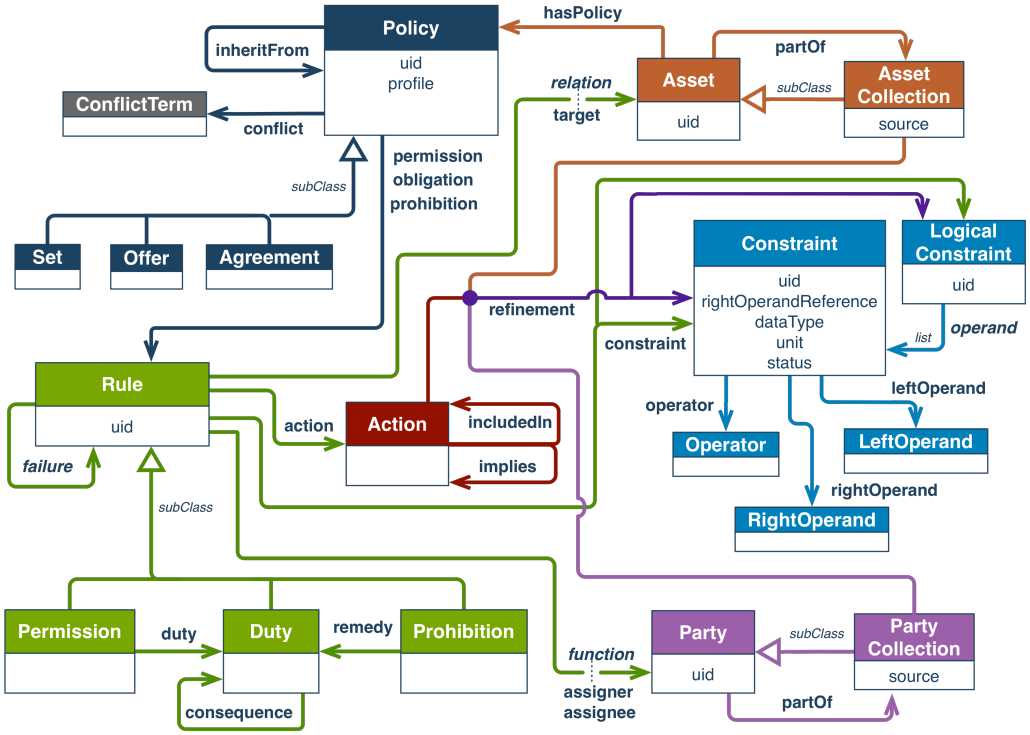


Figure 2.1: ODRL information model [IV18]

completed and the ambitious design requires a dedicated effort from platform designers to be applied. These markets also tend to be isolated, reducing the benefit that derives from the use of a W3C standard. The interoperability is instead a major requirement for the MOSAICrOWN scenario and the use of an existing specification is a significant benefit for the proposal.

We now describe how the MOSAICrOWN policy language can be mapped onto ODRL. In the previous chapter, we have shown that a policy rule consists of four components:

$$\langle (subject, condition), (object, condition), operation, purpose \rangle$$

Considering the broad and flexible ODRL specification model in Figure 2.1 [IV18], the components of the MOSAICrOWN policy rules can be mapped onto the following ODRL entities:

- *subject* corresponds to *assignee*;
- *object* corresponds to *target*;
- *operation* corresponds to *action*;
- *condition* corresponds to *constraint/logical constraint*;
- *purpose* corresponds to *purpose*.

Each of the four entities composing a policy rule in ODRL is uniquely addressed by a *Uniform Resource Identifier* (URI) [Ber], a short string that permits to identify any resource on the web. In this way, any kind of information about each of the entities can be retrieved through subsequent HTTP requests. This also permits the policies to be compliant with the *Linked Data Principles* [W3Cc], as they can provide introspective information by the use of vocabularies or simply by referencing external sources.

```
{
  "@context": [
    "http://www.w3.org/ns/odrl.jsonld",
    "https://www.mosaicrown.eu/ns/mosaicrown.jsonld"
  ],
  "uid": "http://bank.eu/policy/1",
  "permission": [
    {
      "uid": "http://bank.eu/policy/1/1",
      "assignee": "http://bank.eu/user/Alice",
      "target": [
        "http://bank.eu/financial/CardHolder/Name",
        "http://bank.eu/financial/CardHolder/Surname"
      ],
      "action":  "read",
      "purpose": "statistical"
    }
  ]
}
```

Figure 2.2: An example of an ODRL policy expressed in JSON-LD

ODRL specification can be implemented using different serializations: *XML* (*eXtensible Markup Language*), *RDF* (*Resource Description Framework*), *OWL* (*Web Ontology Language*), and *JSON* (*JavaScript Object Notation*). Among these, JSON permits to encode linked data using *JSON-LD* (*JSON for Linked Data*) [W3Ca]. JSON-LD is a way to create a network of standard-based, machine-readable data across web sites. Starting from a single piece of linked data, it is possible to retrieve other linked data that are hosted on different sites across the Web simply following the links to them. This key feature, in addition to the ease of reading and writing, makes JSON-LD the most suitable way to express MOSAICrOWN policies. Figure 2.2 illustrates an example of ODRL policy expressed in JSON-LD. The policy states that `Alice` can `read` attributes `Name` and `Surname` of `CardHolder` for `statistical` purposes.

Note that the use of URIs for identifying the targets in policy rules permits to include any kind of target (i.e., dataset) in the rules. For concreteness and simplicity of description of our policy engine, in this deliverable we will focus on structured data and specifically consider relational databases in our examples. The target of policy rules will then be relations and/or attributes.

Since an official ODRL engine (or processor) [W3Cb] is missing, we have implemented a policy engine processing the MOSAICrOWN policy dialect. Given the importance of this activity from a security point of view, and the need of the engine to access all the policy rules, our implementation is based on the assumption that the policy engine operates in a trusted environment, which is in line with the scenarios considered in MOSAICrOWN.

2.2 Architecture

The goal of the policy engine is to retrieve the MOSAICrOWN policies either from the local file system or the network, import and parse the policies to represent them through a policy knowledge graph (i.e., an entity-relationship graph representing the policy rules), and check whether an access request can be authorized by evaluating it against the knowledge graph. Since the poli-

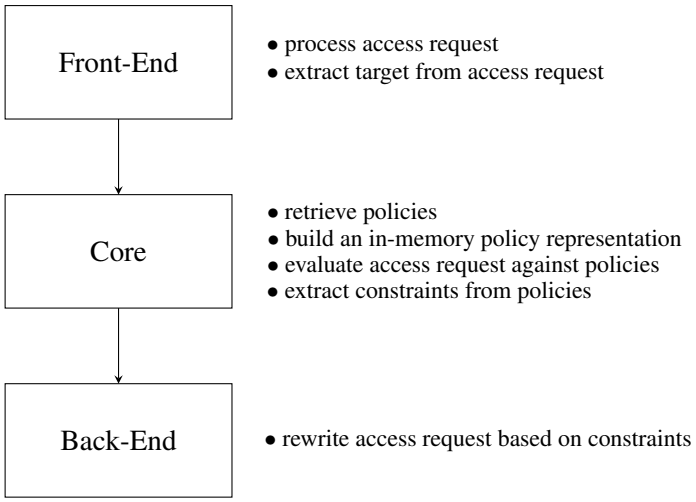


Figure 2.3: Architecture of the policy engine

cies are encoded in ODRL, the implementation of the policy engine started by focusing on the formats available for the ODRL vocabulary [ISMR18a]. According to the ODRL documentation [ISMR18b], policies may be serialized in any supported RDF serialization syntax such as RDF/Turtle, RDF/XML, RDF/N-Triples, and JSON-LD serializations. As already mentioned in the previous Section, for the readability of the policies, we use the JSON-LD serialization.

The implementation of the policy engine has been written in Python due to the availability of libraries that help with the retrieval of policies, the management of knowledge graphs, and the support of multiple serialization standards. In particular, the *RDFLib* library [Tea20] permits to retrieve RDF documents seamlessly, and parse RDF [CWL+14] data written in any supported RDF serialization syntax. In our case, the RDF documents and data are the ODRL ontology, vocabulary, and the MOSAICrOWN policies. By using the RDFLib library, we can directly import the standard ODRL namespaces and vocabularies as well as the specific MOSAICrOWN namespace and vocabulary to introduce the semantic that is not already available in ODRL, such as the purpose component of the policy rules.

To verify whether an access request can be authorized, we need a mechanism for extracting from the policy knowledge graph all policy rules. We then use *SPARQL* [HS13], a semantic query language able to retrieve and manipulate data stored in an RDF graph. SPARQL is integrated in RDFLib and Chapter 3 will describe how SPARQL is used to support the policy engine access decisions. The policy engine can evaluate access requests where the object is represented as an URI. We have also implemented an "extension" to the policy engine to retrieve the object from a query and to enforce the constraints specified in the policy rules. Figure 2.3 shows the policy engine architecture, which is a pipeline composed by the following three modules.

- *Front-End*: it processes an access request in the form of a query and extracts the information about the objects of the access request.
- *Core*: it employs the RDFLib library to verify the policy rules and to make an access decision, which may contain some constraints that need to be enforced.
- *Back-End*: it enforces the constraints received by the Core module by rewriting the query.

Front-End. The Front-End module is responsible for parsing and handling an access request in the form of a query written in a query language. Multiple Front-End modules can be implemented, each one supporting its own query language. The Front-End module retrieves the target `objects` of a query (access request) and complements this information with the information about the `subject` (i.e., the issuer), the kind of `operation` and `purpose` to produce a complete access request that is then passed to the Core module. Our implementation includes two Front-End modules: a module supporting Structured Query Language (SQL), and another module supporting SPARQL (SPARQL Protocol and RDF Query Language).

Core. The Core module contains the main logic of the policy engine. The Core module recovers, processes, and imports the information contained in the ODRL and MOSAICrOWN vocabularies, building an in-memory RDF graph that corresponds to the knowledge graph containing the information needed to understand the MOSAICrOWN policy rules. The Core module then recovers, processes, and imports the MOSAICrOWN policy rules to populate the in-memory knowledge graph representation. The Core component is then ready to process an access request against the knowledge graph to make an access decision that is coherent with the applicable policy rules. The access decision returned by the Core module can be:

- *authorized*: the access request is authorized as-is;
- *conditionally authorized*: the access request has to be modified according to the constraints specified in the policy rules;
- *denied*: the access request is denied.

The Core module is responsible for the retrieval of constraints specified by the policy rules on the objects of the access request to correctly rewrite the access request on the fly.

Back-End. The Back-End module is responsible for the enforcement of the constraints extracted by the Core module. This module then rewrites the original query (access request) by modifying the corresponding Abstract Syntax Tree (AST). Like for the Front-End, different Back-End modules can be implemented, one for each supported query language. We have implemented two modules supporting SQL and SPARQL, respectively.

In the remainder of this deliverable, we describe the three modules composing the architecture of the policy engine more in details. We first describe the Core module (Chapter 3), and then we proceed with the description of the Front-End module(s) (Chapter 4) and the Back-End module(s) (Chapter 5) that complement the working of the policy engine.

3. Core module

In this chapter, we describe the functionality of the Core module of the policy engine, which have been implemented with the objective to realize a flexible policy engine able to support a wider range of use cases. After a brief overview of the working of the Core module (Section 3.1), we illustrate the *joint attribute visibility* property considered in the evaluation of access requests to take access control decisions (Section 3.2). We then describe how the Core module supports the presence of *hierarchies* (Section 3.3) on all components that characterize a policy rule (subject, object, operation, and purpose). Finally, we describe the enforcement of policy constraints (Section 3.4), and the support of prohibition rules (Section 3.5).

3.1 Overview of the Core module

As anticipated in Chapter 2, the Core module performs several tasks related to the parsing and importing of the MOSAICrOWN policy rules in a policy knowledge graph, and to the evaluation of access requests. The construction of the graph is realized by using the RDFLib library [Tea20] that supports the parsing of policies written in RDF [CWL+14] and *JSON-LD* [W3Ca]. *RDFLib* fetches the ODRL vocabulary [ISMR18a] along with the supplementary namespaces and vocabularies referenced by the MOSAICrOWN policy files, directly importing them in the knowledge graph. Immediately after the knowledge graph is available in memory, the Core module can evaluate access requests. The Core module can receive an access request through the Front-End module. The Front-End module structures the access request as a list of quadruples including the assignee, target, action, and purpose. For each quadruple, the Core module first extracts the target objects that, in the relational database context considered in this document (Section 2.1), corresponds to the relational tables and their attributes appearing in the access request. For each target attribute, the Core module then extracts from the knowledge graph all the policy rules that grant visibility to the assignee, for the specific purpose and action of the request. This step is performed through a pre-defined set of *SPARQL* [HS13] queries. SPARQL is a semantic query language for graph data that is directly integrated in RDFLib. Figure 3.1 illustrates an example of SPARQL query that extracts all the MOSAICrOWN rules from a policy.

Policy rules are then grouped by the table to which each target attribute belongs. For each table, the Core module checks whether the joint attribute visibility property (Section 3.2) is satisfied by at least one policy rule. Based on the collected policy rules, the Core module determines whether the access request is:

- permitted, meaning that the joint attribute visibility is satisfied by at least one policy rule;
- conditionally permitted, meaning that the access request (query) must be rewritten considering the additional constraints extracted by the Core module (Chapter 5);
- denied, meaning that the access request is not permitted.

```
PREFIX odrl: <https://www.w3.org/ns/odrl/2/>
PREFIX mosaicrown: <https://www.mosaicrown.eu/ns/mosaicrown/1/>
SELECT DISTINCT ?assignee ?action ?target ?purpose
    WHERE {
        ?policy odrl:permission ?rule .
        ?rule odrl:assignee ?assignee .
        ?rule odrl:action ?action .
        ?rule odrl:target ?target .
        ?rule mosaicrown:purpose ?purpose
    }
```

Figure 3.1: An example of SPARQL query that extracts all rules from a policy

The following sections describe in detail how the Core module enforces the *joint attribute visibility* property, verifies the permission for hierarchical data, extracts access constraints, and how prohibitions are managed.

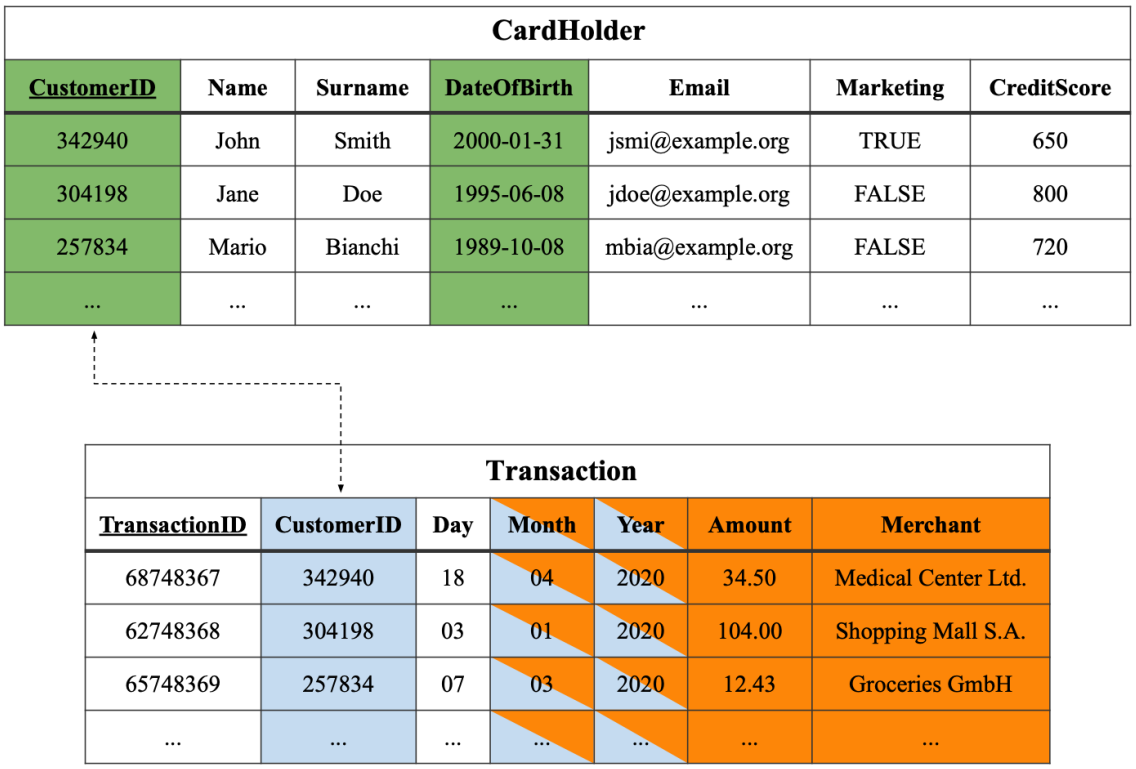
3.2 Joint attribute visibility

A core concept of our model is the *joint attribute visibility* property, which states that if two or more attributes of the same relational table must be accessed together, they must appear together in a policy rule. In other words, the association among different attributes of a same relational table is visible if and only if all attributes appear together in the same policy rule. For instance, if `Alice` can read from a relational table attribute a_1, attribute a_2, or their association (a_1,a_2), we have to define a single policy rule including both attributes in the target component. The definition of two policy rules, one for attribute a_1 and one for attribute a_2, does not have the same effect since `Alice` would not be authorized to access the two attributes together. Note that the joint attribute visibility property applies only to attributes belonging to the same relational table. The access to attributes of different relational tables can instead be regulated by different policy rules. The basic idea is that different policy rules may allow the joint visibility of attributes of different relational tables whenever such policy rules grant access to the attributes that can be used to join the tables, allowing a subject to perform a join operation between the two tables. By imposing this condition for the join operation, we also limit the information that can be obtained by submitting two requests on different relational tables and then performing the join operation at the client side.

As an example, consider the two relational tables in Figure 3.2 and the policy rules in Figure 3.3 defined over the attributes of the two relational tables. These three policy rules allow `Alice` to `read`, for `statistical` purposes, attributes:

- `CustomerID` and `DateOfBirth` of table `CardHolder` (first rule);
- `CustomerID`, `Month`, and `Year` of table `Transaction` (second rule);
- `Month`, `Year`, `Amount`, and `Merchant` of table `Transaction` (third rule).

Assume now that `Alice` submits a request for reading attributes `CustomerID` and `Merchant` of table `Transaction` for `statistical` purposes. This request would be denied because no policy rule allows the joint visibility of these two attributes (`CustomerID` belongs to the target of the second rule and `Merchant` to the target of the third rule). On the other hand, a request over attributes `CustomerID`, `Month`, and `Year` of `Transaction` would be permit-



CardHolder

CustomerID	Name	Surname	DateOfBirth	Email	Marketing	CreditScore
342940	John	Smith	2000-01-31	jsmi@example.org	TRUE	650
304198	Jane	Doe	1995-06-08	jdoe@example.org	FALSE	800
257834	Mario	Bianchi	1989-10-08	mbia@example.org	FALSE	720
...	...	...	...	...	...	...

Transaction

TransactionID	CustomerID	Day	Month	Year	Amount	Merchant
68748367	342940	18	04	2020	34.50	Medical Center Ltd.
62748368	304198	03	01	2020	104.00	Shopping Mall S.A.
65748369	257834	07	03	2020	12.43	Groceries GmbH
...	...	...	...	...	...	...

Figure 3.2: An example of two relational tables with a color-coded representation of the target of the policy rules in Figure 3.3

ted (second rule); the same holds for a request over attributes `Month`, `Year`, and `Merchant` of `Transaction` (third rule). The policy rules defined for `Alice` allow her to perform some analysis on the customers (e.g., count of transactions per card holder per month), and on the merchant (e.g., sum of transaction amount grouped by merchant and month) separately. Furthermore, according to the joint visibility property, these policy rules authorize `Alice` to see the association between attributes `CustomerID` and `DateOfBirth` of table `CardHolder` and attributes `CustomerID`, `Month`, and `Year` of table `Transaction`. In fact, the first and second policy rules have attribute `CustomerID` in common (which is a *foreign key* for table `Transaction`) that can be used to perform a join between the two tables. `Alice` cannot instead access attribute `DateOfBirth` of `CardHolder` together with `Merchant` of `Transaction` as the latter cannot be seen together with the join attribute.

3.3 Hierarchies

The MOSAICrOWN model and language (Chapter 1) supports the definition of hierarchies for all components of a policy rule. Hierarchies simplify the specification of policies and keep their size limited. For instance, suppose that a policy rule should be defined over an entire relational table. Without the support of hierarchies, we have to define a policy rule listing all the URIs of the table attributes. While this does not cause any loss of functionality, it could lead to an unnecessary large policy when dealing with tables with many attributes. We have therefore extended the first version of the policy engine described in Deliverable D3.3 [DS20] with the support of hierarchies.

```
{
  "@context": [
    "https://www.w3.org/ns/odrl.jsonld",
    "https://www.mosaicrown.eu/ns/mosaicrown.jsonld"
  ],
  "uid": "http://bank.eu/policy/1",
  "permission": [
    {
      "uid": "http://bank.eu/policy/1/1",
      "assignee": "http://bank.eu/user/Alice",
      "target": [
        "http://bank.eu/financial/CardHolder/CustomerID",
        "http://bank.eu/financial/CardHolder/DateOfBirth"
      ],
      "action": "read",
      "purpose": "statistical"
    },
    {
      "uid": "http://bank.eu/policy/1/2",
      "assignee": "http://bank.eu/user/Alice",
      "target": [
        "http://bank.eu/financial/Transaction/CustomerID",
        "http://bank.eu/financial/Transaction/Month",
        "http://bank.eu/financial/Transaction/Year"
      ],
      "action": "read",
      "purpose": "statistical"
    },
    {
    "uid": "http://bank.eu/policy/1/3",
    "assignee": "http://bank.eu/user/Alice",
    "target": [
      "http://bank.eu/financial/Transaction/Month",
      "http://bank.eu/financial/Transaction/Year",
      "http://bank.eu/financial/Transaction/Amount",
      "http://bank.eu/financial/Transaction/Merchant"
    ],
    "action": "read",
    "purpose": "statistical"
    }
  ]
}
```

Figure 3.3: An example of a policy with three rules

3.3.1 Data category hierarchy

ODRL supports the *partOf* property that can be used for defining a data category hierarchy. Intuitively, the target of a policy rule can be defined as *part of* an entity, thus establishing a hierarchical relationship between the target and the specified entity. For instance, Figure 3.4 illustrates how the *partOf* property can be used to indicate that attribute `Name` in the target is part of the `CardHolder` table. Figure 3.5 shows the RDF graph corresponding to the *partOf* property in Figure 3.4.

```
{
  "@type": "odrl:target",
  "@id":"http://bank.eu/financial/CardHolder/Name",
  "odrl:partOf": "http://bank.eu/financial/CardHolder"
}
```

Figure 3.4: An example of the use of the *partOf* property in JSON-LD format

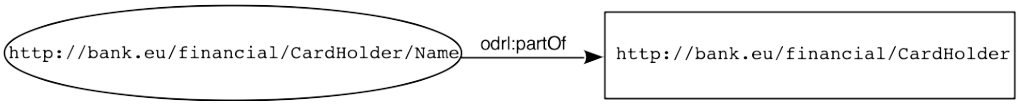


Figure 3.5: RDF graph corresponding to the definition in Figure 3.4

```
{ "@context":
    ["http://www.w3.org/ns/odrl.jsonld","http://localhost:8000/ns/
       mosaicrown/namespace.jsonld"],
  "@type":"Set", "uid": "http://bank.eu/policy/p1",
  "permission": [
        { "uid": "http://bank.eu/policy/p1_perm_1",
          "assignee": "http://bank.eu/user/administrative",
          "target": "http://bank.eu/financial/CardHolder",
          "action": "read",
          "purpose": "statistical"
        },
        { "uid": "http://bank.eu/policy/p1_perm_2",
          "assignee": "http://bank.eu/user/administrative/agentA",
          "target": "http://bank.eu/financial/CardHolder/Name",
          "action": "write",
          "purpose": "institutional"
        }]
}
```

Figure 3.6: An example of policy containing hierarchies

The main problem with this manual specification of the *partOf* property inside an ODRL policy is that it increases the overall policy size. To avoid this, we leverage the intrinsic hierarchical information that is enclosed in the concept of URI [Ber]. We have therefore added a step in the construction of the knowledge graph that expands the graph through the application of the following procedure.

- Each RDF node representing a *target* is retrieved from the knowledge graph.
- For each node retrieved, every possible subpath of the corresponding URI is derived.
- For each subpath, the RDF triplet $\langle$*subpath*, *odrl:partOf*, *subpath_parent*$\rangle$ is added to the knowledge graph.

As an example, consider the policy in Figure 3.6. After the initial parsing of the policy, the corresponding RDF knowledge graph contains two targets: `http://bank.eu/financial/CardHolder` and `http://bank.eu/financial/CardHolder/Name`. By applying the procedure abovementioned, the knowledge graph is enriched with the following RDF triplets:

- ⟨*http://bank.eu/financial*, *odrl:partOf*, *http://bank.eu*⟩
- ⟨*http://bank.eu/financial/CardHolder*, *odrl:partOf*, *http://bank.eu/financial*⟩
- ⟨*http://bank.eu/financial/CardHolder/Name*, *odrl:partOf*, *http://bank.eu/financial/CardHolder*⟩

This expansion of the knowledge graph permits the Core module to correctly interpret (and possibly take an access decision) the hierarchical structure of the targets specified in the policy. Consider again the policy in Figure 3.6. The RDF triplets added to the knowledge graph allow the Core module to interpret the two rules in the policy as rules regulating:

- a read access to the entire `CardHolder` table for `statistical` purposes (this is done without listing all attributes of the table);
- a write access only to the `Name` attribute of the `CardHolder` table for `institutional` purposes.

3.3.2 Subject, operation, and purpose hierarchies

Since subject and operations can be directly expressed in ODRL with the *assignee* and *action* constructs, we can also leverage the existing constructs of ODRL for expressing a hierarchical relationship between subjects and operations. In particular, we can use:

- *odrl:partOf* to express a hierarchical relationship between two assignees;
- *odrl:includedIn* to express a hierarchical relationship between two actions.

However, since *odrl:partOf* is already used for the target component, we have introduced a new predicate to express the hierarchical relationship among assignees. The main advantage of having different predicates for the target and the assignee is that we can build a knowledge graph that models the hierarchical relationships of targets and assignees without ambiguity. The new predicate, called *mosaicrown:belongsTo*, is defined in the *mosaicrown vocabulary* using OWL [Gro12]. Also, the purpose hierarchy requires to introduce a new predicate, called *mosaicrown:declinationOf*, in the *mosaicrown vocabulary* since the purpose component is not directly supported by ODRL. Note that the procedure that expands the knowledge graph with the information about the data category hierarchy can also be applied to the assignee component. For instance, consider the policy in Figure 3.6. The expansion procedure first retrieves from the knowledge graph all the objects referenced by the *odrl:assignee* predicate. Then, it adds the following RDF triplets to the graph:

- ⟨*http://bank.eu/user*, *mosaicrown:belongsTo* , *http://bank.eu*⟩
- ⟨*http://bank.eu/user/administrative*, *mosaicrown:belongsTo*, *http://bank.eu/user*⟩
- ⟨*http://bank.eu/user/administrative/agentA*, *mosaicrown:belongsTo*, *http://bank.eu/user/administrative*⟩

After this expansion step, the knowledge graph contains information about:

- a rule that authorizes read access to all users in the `administrative` group;
- a second rule that authorizes write access to the `agentA` user who, being a member of the `administrative` group, has also the privilege granted by the previous rule.

Note that the expansion procedure used for the target and assignee components of a policy rule does not apply to the action and purpose components. In fact, the action and purpose hierarchies must be explicitly specified in custom vocabularies defined by the data owner and their content is included into the knowledge graph when policy files are parsed. Figure 3.7 and Figure 3.8 show

```
{ "@id": "http://localhost:8000/ns/mosaicrown/read",
  "@type": [
    "http://www.w3.org/ns/odrl/2/action",
    "http://www.w3.org/2004/02/skos/core#Concept" ],
  "http://www.w3.org/2000/01/rdf-schema#isDefinedBy":
    [{"@id": "http://localhost:8000/ns/mosaicrown/"}],
  "http://www.w3.org/2000/01/rdf-schema#label":
    [{ "@language": "en", "@value": "Read" }],
  "http://www.w3.org/2004/02/skos/core#definition":
    [{ "@language": "en", "@value": "The permission to read a target"
       }],
  "http://www.w3.org/ns/odrl/2/includedIn":
    [{"@id":"http://localhost:8000/ns/mosaicrown/write" }]
}
```

Figure 3.7: An example of the definition of an action hierarchy in a vocabulary

```
{ "@id": "http://localhost:8000/ns/mosaicrown/institutional",
  "@type": [ "http://localhost:8000/ns/mosaicrown/purpose",
             "http://www.w3.org/2004/02/skos/core#Concept" ],
  "http://www.w3.org/2000/01/rdf-schema#isDefinedBy":
    [{ "@id": "http://localhost:8000/ns/mosaicrown/"}],
  "http://www.w3.org/2000/01/rdf-schema#label":
    [{ "@language": "en", "@value": "institutional" }],
  "http://www.w3.org/2004/02/skos/core#definition":
    [{ "@language": "en", "@value": "An institutional purpose" }],
  "http://localhost:8000/ns/mosaicrown/declinationOf":
    [{"@id": "http://localhost:8000/ns/mosaicrown/general"}]
}
```

Figure 3.8: An example of the definition of a purpose hierarchy in a vocabulary

an example of the definition of an action hierarchy and a purpose hierarchy, respectively. More precisely, Figure 3.7 shows the definition of an action hierarchy where the `read` action is *included in* the `write` action. This implies that a policy rule authorizing a `write` privilege will also authorize the `read` privilege. Figure 3.8 shows the definition of a purpose hierarchy where purpose `institutional` is a *declination of* the `general` purpose. This implies that a rule authorizing an access for the `general` purpose will also authorize an access for the `institutional` purpose.

3.3.3 Construction of the knowledge graph

The introduction of hierarchies over the target, assignee, action, and purpose components requires a change in the SPARQL query used to retrieve the policy rules needed to evaluate an access request. Figure 3.9 shows the new query. This query can be used to recursively retrieve the policy rules after the initialization of the *predicate*, *action*, *target*, *assignee*, and *purpose* variables. Specifically:

- **predicate**: this variable can be either equal to *odrl:permission* or *odrl:prohibition* (see Sec-

```
PREFIX odrl: <https://www.w3.org/ns/odrl/2/>
PREFIX mosaicrown: <https://www.mosaicrown.eu/ns/mosaicrown/1/>
    SELECT DISTINCT ?rule
    WHERE {
      ?policy ?predicate ?rule .
      ?rule odrl:assignee ?assigneeRec .
      ?assignee mosaicrown:belongsTo* ?assigneeRec .
      ?rule odrl:action ?actionRec .
      ?action odrl:includedIn* ?actionRec .
      ?rule odrl:target ?targetRec .
      ?target odrl:partOf* ?targetRec .
      ?rule mosaicrown:purpose ?purposeRec .
      ?purpose mosaicrown:declinationOf* ?purposeRec .
   }
```

Figure 3.9: SPARQL query used to retrieve the policy rules inside a policy

tion 3.5);

- **action**: this variable contains the URI of the operation specified in the access request;
- **target**: this variable contains the URI of the dataset specified in the access request;
- **assignee**: this variable contains the URI of the subject that is submitting the access request;
- **purpose**: this variable contains the URI of the purpose specified in the access request.

Note that the new SPARQL query is also compatible with policies that do not use hierarchies. For instance, the following statement included in the query in Figure 3.9:

?assignee mosaicrown:belongsTo ?assigneeRec*

retrieves the RDF nodes reachable starting from a node marked *assignee* and following zero or more *mosaicrown:belongsTo* predicate path. To increase the performance of the policy engine, we also deferred the expansion of the knowledge graph. This permits to reduce the time required for its creation. Subject and target expansions are then performed on-the-fly when an access request is submitted to the policy engine. To explain the on-the-fly expansion, consider the policy in Figure 3.6. The following access request:

⟨`user.administrative.agentA, CardHolder.BirthDate, read, statistical`⟩

can be evaluated only after the execution of the expansion procedure on the target component that has the effect of adding the following RDF triplet to the knowledge graph:

⟨*http://bank.eu/financial/CardHolder/BirthDate, odrl:partOf, http://bank.eu/financial/CardHolder*⟩

The same observation applies to the assignee component. Again, consider the policy in Figure 3.6 and the following two access requests:

⟨`user.advertisement.agentB, CardHolder.Name, write, institutional`⟩

⟨`user.advertisement.agentC, CardHolder.Name, write, institutional`⟩

After the execution of the expansion procedure over the assignee/subject component of the policy rules and access requests, the knowledge graph contains the hierarchical information shown in Figure 3.10. In this case, the two access requests are denied, since `agentB` and `agentC` do not belong to the `administrative` group.

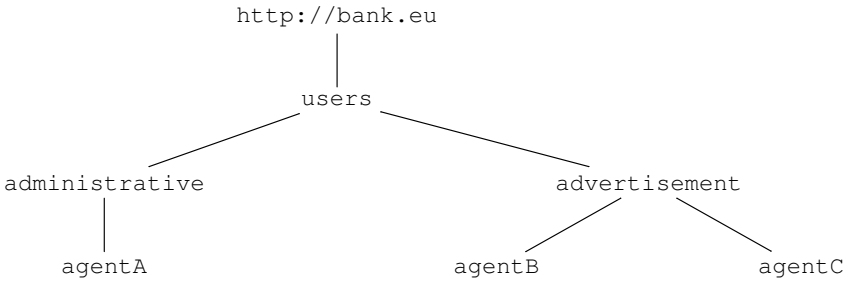


Figure 3.10: Subject hierarchy represented in the RDF knowledge graph after the on-the-fly expansion

3.4 Constraints

The MOSAICrOWN policy rules can specify *conditions* defining constraints that can be of three different types:

- conditions on the content of datasets;
- conditions on subject profiles and metadata;
- conditions on location and time of an access request.

Since we aim at maintaining a seamless integration with the existing constructs of ODRL, conditions are implemented leveraging the *Constraint* and *Logical Constraint* classes of ODRL. Class *Constraint* is used to indicate one condition that must be satisfied to consider the rule valid. Multiple instances of *Constraint* in the same policy rule are ANDed, meaning that all conditions must be satisfied to consider the rule valid. Figure 3.11 illustrates an example of policy rule with a single instance of class *Constraint*. The rule states that the `administrative` users can read for `statistical` purposes the tuples in `CardHolder` with a `CreditScore` equals to 650.

The *Logical Constraint* class is used to specify a composition of two or more constraints with a logical operator. Figure 3.12 illustrates an example of policy rule stating that the `administrative` users can read for `statistical` purposes the tuples in `CardHolder` with a `CreditScore` greater than or equal to 800 OR with `Marketing` equals to TRUE.

Note that any condition on the content of datasets (e.g., the conditions in the policy rules reported in Figures 3.11 and 3.12) can be directly evaluated by the Core module. In this case, such conditions are combined through the proper logical operator and are then sent to the Back-End module. The Back-End module then rewrites the access request by including the conditions received by the Core module (Chapter 5). Conditions on the profile of subjects and on time or location of an access request cannot instead be directly evaluated by the Core module. In this case, the Core module has to interact with other components. For instance, suppose that we define a policy rule stating that an access to a specific dataset can be granted only to the European citizens. To evaluate this rule, the Core module needs to access the subject profile that has to be provided by the data market platform.

```
{
 "@context": [
   "http://www.w3.org/ns/odrl.jsonld",
   "http://localhost:8000/ns/mosaicrown/namespace.jsonld"
 ],
 "@type": "Set",
 "uid": "http://bank.eu/policy/p2",
 "permission": [{
   "uid": "http://bank.eu/policy/p2_perm_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder"],
   "action": "read",
   "purpose": "statistical",
   "constraint": [{
      "leftOperand": "http://bank.eu/financial/CardHolder/CreditScore",
      "operator": "eq",
      "rightOperand":  "650"
   }]
 }]
}
```

Figure 3.11: An example of policy rule containing a constraint

```
{
 "@context": [
   "http://www.w3.org/ns/odrl.jsonld",
   "http://localhost:8000/ns/mosaicrown/namespace.jsonld"
  ],
 "@type": "Set",
 "uid": "http://bank.eu/policy/p2",
 "permission": [{
   "uid": "http://bank.eu/policy/p2_perm_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder"],
   "action": "read",
   "purpose": "statistical",
   "constraint": {
       "or":[{
        "leftOperand": "http://bank.eu/financial/CardHolder/CreditScore",
        "operator": "gteq",
        "rightOperand":  "800"
      },{
        "leftOperand": "http://bank.eu/financial/CardHolder/Marketing",
        "operator": "eq",
        "rightOperand":  "TRUE"
      }]
   }
 }]
}
```

Figure 3.12: An example of policy rule containing a logical constraint

```
{
 "@context": [
 "http://www.w3.org/ns/odrl.jsonld",
 "http://localhost:8000/ns/mosaicrown/namespace.jsonld"
 ],
  "@type": "Set",
  "uid": "http://bank.eu/policy/p3",
  "permission": [{
   "uid": "http://bank.eu/policy/p3_perm_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder/"],
   "action": "read",
   "purpose": "statistical"
  }],
 "prohibition": [{
   "uid": "http://bank.eu/policy/p3_proh_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder/Marketing"],
   "action": "read",
   "purpose": "statistical"
 }]
}
```

Figure 3.13: An example of policy containing a prohibition

3.5 Prohibitions

To improve the flexibility of the MOSAICrOWN language, we have introduced the support for *prohibitions*. A prohibition allows a data owner to explicitly indicate what must not be done with a dataset inside the data market. To add this functionality, we leveraged the existing concept of *prohibition* in ODRL. Similar to the *permission* construct, *prohibition* is an extension of the *rule* construct and shares the same structure, as it is visible in the example of Figure 3.13. In this example, the policy includes two rules: a permission stating that the `administrative` users can read for `statistical` purposes the content of table `CardHolder`; a prohibition stating that the `administrative` users cannot read for `statistical` purposes attribute `Marketing` of table `CardHolder`. Intuitively, the combination of these two rules has the effect that the `administrative` users can read all attributes of the `CardHolder` table but attribute `Marketing` for `statistical` purposes. The implementation of prohibitions requires to adapt: *1)* the method used to retrieve the policy rules that apply to the target of an access request; and *2)* the method used to evaluate whether an access request can be authorized. With respect to the first aspect, we have already shown that variable *predicate* in the SPARQL query in Figure 3.9 allows the retrieval of permissions (value *odrl:permission*) or prohibitions (value *odrl:prohibition*). With respect to the second aspect, the process for checking whether an access request can be authorized is modified. Intuitively, both permissions and prohibitions that apply to an access request are retrieved from the knowledge graph. Since we assume that prohibitions take precedence over permissions, if there exists a prohibition, the access request is denied. Otherwise, if there exists a permission, the access request can be authorized or conditionally authorized as detailed in the following.

- Both prohibitions and permissions are first retrieved from the knowledge graph using two SPARQL queries.
- Permissions and prohibitions are then grouped in two separate rule sets.
- Prohibitions are first evaluated. If there is at least one prohibition, the access request is denied. Otherwise, the process continues with the evaluation of permissions.
- Permissions are evaluated. If there is at least one permission that is compliant with the joint visibility property, the access request is granted or conditionally granted when there are constraints on the object appearing in the access request. Otherwise, the access request is denied.

Note that also prohibitions support the definition of hierarchies on all components of the policy rule. Furthermore, they do not need to satisfy the joint visibility property: a prohibition applies to an access request if at least one of its targets appears in the access request.

4. Front-End module

In this chapter, we describe the Front-End modules implemented. The main goal of a Front-End module is to parse an access request written using a query language to construct the corresponding Abstract Syntax Tree (AST), and to extract the targets of the request. Clearly, since different query languages operate according to different paradigms, different Front-end modules have to be implemented, one for each query language that needs to be supported. Each Front-End module receives an access request written according to the supported query language as input, and produces a list of quadruples including the assignee, target, action, and purpose as output. These quadruples are then passed to the Core module. Note, in particular, that the Front-End provides a mapping between the targets of an access request and their URIs. As already anticipated in Chapter 2, we have implemented two Front-End modules: a module supporting SQL and relational tables (Section 4.1), and a module supporting SPARQL and RDF knowledge graphs (Section 4.2).

4.1 Structured Query Language (SQL)

The Front-End module realized for SQL parses the input query through *sqlparse* [Alb], a non-validating SQL parser for Python that also creates the corresponding *Parse Tree* (*PT*) representation. The Parse Tree is a concrete representation of the input that preserves all the information, formatting included. For instance, the application of *sqlparse* to the SQL query in Figure 4.1(a) produces the Parse Tree in Figure 4.1(b).

As it is visible from the example of query, we use fully qualified names for attributes (e.g., `CardHolder.CustomerID`) to avoid collisions of names. This allows our Front-End module to extract the necessary information from the query without the need to have previous knowledge of the relation schema, and guarantees uniqueness in the representation of the query targets.

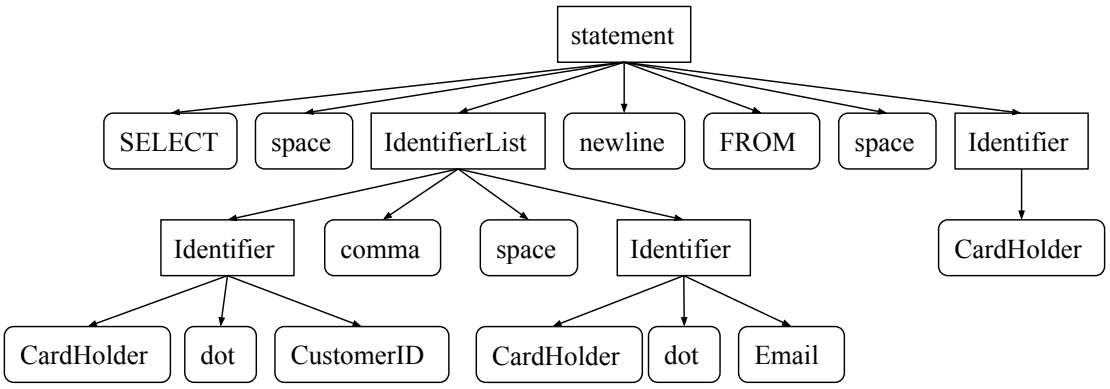
Once the Front-End module has obtained the Parse Tree representation of the query, it traverses the tree to discover the query targets. The tree traversal keeps track of the relational tables involved in the query (exploring the comma separated list in the `FROM` clause), and identifies the attributes in the:

- comma separated list in the `SELECT` clause;
- expressions following the `ON` keyword and in the `WHERE`, and `HAVING` clauses;
- comma separated lists in the `GROUP BY` and `ORDER BY` clauses;
- comma separated lists within brackets following the name of a function (function parameters);
- structured statements that extensively use expressions (`CASE` clause).

This information is represented in an Abstract Syntax Tree (AST), which is a tree representing the abstract syntactic structure of the query. The root node represents the query itself, while its

```
SELECT CardHolder.CustomerID, CardHolder.Email
FROM CardHolder
```

(a)



(b)

Figure 4.1: An example of SQL query (a) and corresponding Parse Tree obtained through the execution of *sqlparse* (b)

child nodes represent its sub-queries. The tool supports sub-queries in the `SELECT`, `ON`, `WHERE`, and `HAVING` clauses, in the list of tables expressions, and in the function parameter. The AST supports the identification of the scope of tables and aliases, facilitating the understanding of the overall query structure.

Although the Front-End module understands the query structure, when the module provides the targets of the access request to the subsequent steps of the pipeline, the query is considered as a whole, without splitting the accessed attributes by sub-query. While too restrictive, this strategy prevents circumventions of the join attribute visibility property that can happen by abusing the query syntax to make attributes appear in different sub-queries. Our solution therefore does not verify how the query is written but only operates on the accessed data. This implies that, even if a query is actually composed of two (or more) queries that do not join their data, our approach will be over-restrictive, as the attributes are considered all together. However, since such a query could be split into two queries by the requester, we are not limiting the utility.

Once all targets have been extracted from the query, they are mapped to their URIs to be processed by the Core module. This is done by providing a mapping from the tables to their URIs, and then by appending the name of the attributes of the table to their corresponding URI segment. Figure 4.2 shows a mapping of table names to URIs, and the corresponding URI representation of the targets of the query in Figure 4.1.

As already discussed, Figure 4.2 shows that the adoption of fully qualified names permits to map different attributes with the same name that belong to the schema of different tables (e.g., `CustomerID`) to exactly one URI, removing any ambiguity on the target of the query.

Table	URI
`CardHolder`	http://bank.eu/financial/CardHolder
`Transaction`	http://bank.eu/financial/Transaction

(a)

Table	URI
`CardHolder`	http://bank.eu/financial/CardHolder

Attribute	URI
`CustomerID`	http://bank.eu/financial/CardHolder/CustomerID
`Email`	http://bank.eu/financial/CardHolder/Email

(b)

Figure 4.2: An example of mapping of table names to URIs (a) and of URI representation of the targets of the query in Figure 4.1 (b)

```
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
SELECT ?professor ?city ?age WHERE {
   ?professor a dbpedia-owl:Professor.
   ?professor dbpedia-owl:birthPlace | dbpedia-owl:residence ?city.
   ?professor dbpedia-owl:age ?age
   FILTER (?professor IN (<http://example.org/Alice>,
      <http://example.org/Bob>)).
   FILTER (?age < 45)
}
```

Figure 4.3: An example of SPARQL query that can be parsed by the policy engine

4.2 SPARQL Protocol and RDF Query Language (SPARQL)

The Front-End module implemented for SPARQL parses an access request through the RDFLib library [Tea20], which allows the Front-End module to operate on the Abstract Syntax Tree corresponding to the parsed query. Since SPARQL has a more complex query structure than SQL, our implementation focuses on the extraction of all the triplets (subject, predicate, object) from the `WHERE` statement, and of all the `FILTER` statements of the form (subject, operator, value). As an example, consider the query in Figure 4.3. In this case, the Front-End module will retrieve the following triplets/filters.

- Three triplets, corresponding to the first three lines of the `WHERE` statement. Note that the Front-End module is able to parse a triplet even if multiple alternatives for the `predicate` are specified through the `|` SPARQL operator.
- Two filters, corresponding to the last two lines of the `WHERE` statement. Note that several types of `FILTER` statements are supported:
  - comparison with a literal value;
  - comparison with a node Value;

- `IN` statements with a single or multiple values specified.

The example also shows that the parser is able to correctly handle the `PREFIX` statement. This permits to guarantees uniqueness in the representation of the query targets in a similar way to what is done with fully qualified names for the SQL Front-End.

5. Back-End module

In this chapter, we describe the Back-End modules implemented. The goal of a Back-End module is to enforce the constraints extracted by the Core module. This enforcement is realized by rewriting an access request in a way that depends on the language used for writing the request itself. In the following, we first describe how the constraints that need to be enforced by the Back-End module are extracted from the knowledge graph (Section 5.1), and then illustrate the implemented modules, that is, a Back-End module for SQL (Section 5.2) and a Back-End module for SPARQL (Sections 5.3).

5.1 Constraints discovery

The knowledge graph processed by the Core module can also be used for extracting the constraints associated with a policy rule. Figure 5.1 shows the SPARQL query used for extracting these constraints. As it is visible from the figure, the query is composed by two parts, corresponding to the two `WHERE` statements. The first `WHERE` statement handles policy rules that include a single instance or multiple instances of class *constraint* (e.g., see Figure 3.11). The second `WHERE` statement handles policy rules that include constraints combined with the *odrl:and* operator or *odrl:or* operator (e.g., see Figure 3.12).

Note that in the two `WHERE` statements the *target* variable is initialized with the URI corresponding to the target that is being accessed by the access request (query). For instance, a valid URI could be *http://bank.eu/financial/CardHolder*, which is a possible output of a Front-End module. The tuples recovered from the SPARQL query in Figure 5.1 contain the following fields.

- **leftOperand**: URI of the target that is constrained.
- **operator**: URI of the operator used to compare the *leftOperand* with the *rightOperand*. The admitted values are detailed in the following sections that describe the implementation of the Back-End modules.
- **rightOperand**: URI or terminal value compared to the *leftOperand*.
- **type**: URI of the rule type (*odrl:permission* or *odrl:prohibition*).
- **logOperator**: URI of the logical operator used in the constraint. This field is used only in *logical constraints*, and is *undefined*, otherwise.
- **logcon**: URI of the logical constraint retrieved. This field is used only in *logical constraints*, and is *undefined*, otherwise.

5.2 Structured Query Language (SQL)

The Back-End module realized for SQL supports the following operators:

```
SELECT DISTINCT
   ?leftOperand ?operator ?rightOperand ?type ?logOperator ?logcon
WHERE {
   {
      ?rule odrl:target ?target.
      ?policy ?type ?rule.
      ?rule odrl:constraint ?con.
      ?con odrl:leftOperand ?leftOperand.
      ?con odrl:operator ?operator.
      ?con odrl:rightOperand ?rightOperand
   } UNION
   {
       ?rule odrl:target ?target.
       ?policy ?type ?rule.
       ?rule odrl:constraint ?logcon.
       ?logcon ?logOperator ?con.
       ?con odrl:leftOperand ?leftOperand.
       ?con odrl:operator ?operator.
       ?con odrl:rightOperand ?rightOperand
       FILTER (?operand IN (odrl:and, odrl:or))
   }
}
```

Figure 5.1: SPARQL query used for extracting constraints from policy rules

- **odrl:eq** corresponds to the "=" SQL operator;
- **odrl:gt** corresponds to the ">" SQL operator;
- **odrl:gteq** corresponds to the ">=" SQL operator;
- **odrl:lt** corresponds to the "<" SQL operator;
- **odrl:lteq** corresponds to the "<=" SQL operator;
- **odrl:neq** corresponds to the "!=" SQL operator;

When searching for the constrained target through the query shown in Figure 5.1 the *target* variable is initialized with an URI representing the relation attribute that is being accessed. For instance, given the query: `SELECT CardHolder.CustomerID FROM CardHolder`, the target variable is mapped to `http://bank.eu/financial/CardHolder/CustomerID` once processed by the Front-End component. Constraints in policy rules are represented according to the following representation:

- a single triplet of the form $(attribute, operator, value)$ represents a single constraint;
- a list of triplets of the form $(attribute, operator, value)$ represents multiple constraints combined in OR;
- multiple lists of triplets of the form $(attribute, operator, value)$ represent a conjunction (AND) of multiple constraints combined in OR.

As an example, consider the policies in Figure 5.2 and Figure 5.3. (Note that these policies have been already illustrated in Figure 3.11 and Figure 3.12 and they have been duplicated here for the sake of readability.)

```
{
 "@context": [
   "http://www.w3.org/ns/odrl.jsonld",
   "http://localhost:8000/ns/mosaicrown/namespace.jsonld"
 ],
 "@type": "Set",
 "uid": "http://bank.eu/policy/p2",
 "permission": [{
   "uid": "http://bank.eu/policy/p2_perm_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder"],
   "action": "read",
   "purpose": "statistical",
   "constraint": [{
      "leftOperand": "http://bank.eu/financial/CardHolder/CreditScore",
      "operator": "eq",
      "rightOperand":  "650"
   }]
 }]
}
```

Figure 5.2: An example of policy rule containing a constraint

```
{
 "@context": [
   "http://www.w3.org/ns/odrl.jsonld",
   "http://localhost:8000/ns/mosaicrown/namespace.jsonld"
  ],
 "@type": "Set",
 "uid": "http://bank.eu/policy/p2",
 "permission": [{
   "uid": "http://bank.eu/policy/p2_perm_1",
   "assignee": "http://bank.eu/user/administrative",
   "target": ["http://bank.eu/financial/CardHolder"],
   "action": "read",
   "purpose": "statistical",
   "constraint": {
       "or":[{
        "leftOperand": "http://bank.eu/financial/CardHolder/CreditScore",
        "operator": "gteq",
        "rightOperand":  "800"
      },{
        "leftOperand": "http://bank.eu/financial/CardHolder/Marketing",
        "operator": "eq",
        "rightOperand":  "TRUE"
      }]
   }
 }]
}
```

Figure 5.3: An example of policy rule containing a logical constraint

The constraint in the first policy is represented as [(`CardHolder.CreditScore`, =, 650)], and the constraint in the second policy is represented as [(`CardHolder.CreditScore`, >=, 800), (`CardHolder.Marketing`, =, TRUE)]. Once all constraints have been retrieved and represented as described above, the last step of the rewriting process can be executed. In particular, the Back-End module performs the following tasks.

- All occurrences of the considered target are retrieved using the tree representation produced by the Front-End module.
- Each occurrence of the target is substituted by a subquery enforcing the constraints that must be satisfied when accessing the target.
- For each substitution, all nodes needed for defining an alias (i.e., whose name matches the one used in the query for the original table) are added to the tree.

As an example, consider again the policy in Figure 5.3 and the query `SELECT CardHolder.Name, CardHolder.Surname FROM CardHolder`. The rewriting process produces the following query:

```
SELECT C.Name, C.Surname
FROM (
  SELECT *
  FROM CardHolder
  WHERE ((CreditScore >= 800  OR Marketing = TRUE))
) AS C
```

Note that the rewriting process is not limited to one target or to the `FROM` statement, but can also be used for:

- enforcing constraints on multiple tables;
- rewriting multiple occurrences of the same table in distinct parts of the query;
- rewriting occurrences of the target table located inside sub-queries;
- rewriting occurrences of the target table even when an alias is applied. In this case the alias defined for the target table will be associated with the added subqueries.

5.3 SPARQL Protocol and RDF Query Language (SPARQL)

The Back-End module implemented for enforcing constraints on SPARQL queries takes the triplets generated by the Front-End module as input. As described in Section 4.2, each triplet is composed by a subject, predicate, and an object. If the predicate in a triplet matches the target on which a policy rule defines a constraint, the corresponding subject is extracted from the triplet. For each pair (subject, predicate), a new triplet and a `FILTER` predicate are created to enforce the constraint. These elements have the following form:

- `?recovered_subject constrained_target dummy_variable`
- `FILTER(dummy_variable constraint_operator constraint_value)`

As an example, consider the policy in Figure 5.2 and the following SPARQL query submitted by an `administrative` user (http://bank.eu/user/administrative).

```
SELECT ?Name
WHERE {
    ?CardHolder a <http://bank.eu/financial/CardHolder>.
    ?CardHolder <http://www.w3.org/2006/vcard/ns\#Name> ?Name
}
```

The policy engine performs the following steps:

1. the triplet

   $\langle ?\textit{CardHolder}, \texttt{rdf:type}, \texttt{http://bank.eu/financial/CardHolder} \rangle$

   is recovered from the AST by the *RDFLib* parser, since the predicate `http://bank.eu/financial/CardHolder` is the target of a policy with a constraint;
2. the **CardHolder** variable is stored with its corresponding predicate;
3. a new dummy RDF variable, named **x**, is created;
4. the RDF triplet

   $\langle ?\textit{CardHolder}, \texttt{http://bank.eu/financial/CardHolder/CreditScore}, ?x \rangle$

   is created and the statement needed for its retrieval is added to the AST of the query;
5. the `FILTER` statement `FILTER(?x = "650")` is added to the AST of the query.

The produced AST can then be processed by a SPARQL query engine and the result will be equal to the one produced by the query:

```
SELECT ?Name
WHERE {
   ?CardHolder  a <http://bank.eu/financial/CardHolder>.
   ?CardHolder <http://www.w3.org/2006/vcard/ns#Name> ?Name.
   ?CardHolder <http://bank.eu/financial/CardHolder/CreditScore> ?x
   FILTER (?x  = "650")
}
```

In this case, the query will return only the RDF nodes that are linked to the `650` RDF Literal through the predicate `http://bank.eu/financial/CardHolder/CreditScore`; each `CardHolder` node without this kind of link or with the wrong literal will be then removed from the result set of the query.

6. Conclusions

In this deliverable, we have described the policy engine tool developed in the context of Work Package 3. Work Package 3 has addressed the problem of defining a data governance framework for managing data and for specifying policies in the data market context. This problem has been approached by defining a model and a declarative policy language that can be used for specifying, in a flexible way, different protection requirements that may need to be imposed on data. The policy engine tool has been implemented for enforcing such policies. The policy engine has been designed to be compatible with existing technology, thus resulting deployable in real systems. The policy engine tool has been written in Python and builds on the Open Digital Rights Language (ODRL), a W3C policy specification standard based on XML. A first version of the policy model and language as well as of the policy engine have been presented in Deliverable D3.3 "First version of policy specification language and model" [DS20]. In this deliverable, we have described the new features introduced in the policy engine. Chapter 1 has provided an overview of the main concepts of the policy model and language. While simple, the model results expressive, with support for abstractions on the different elements characterizing a policy rule, conditions on metadata and subjects' profiles, as well as explicit consideration of purpose of use, thus responding to the needs of the use cases and to data privacy regulations.

Chapter 2 has presented how the high-level policy specifications can be encoded in ODRL, and then has described the architecture of the policy engine tool. The policy engine is composed by three components (i.e., Front-End, Core, and Back-End) and takes an access request written according to a specific query language (Front-End), checks the access request against the policies, and returns an access decision (Core). The access decision can authorize the access request as it is, can require a rewrite of the access request (Back-End), or can be denied. The goal in the design and implementation of the policy engine was to provide a tool that provides efficient policy evaluation and enjoys interoperability with current technology.

Chapter 3 has provided a description of the Core module. The features supported by the Core model are related to the verification of the joint attribute visibility property, and the management of hierarchies on all the components of a policy rule (i.e., subject, object, operation, and purpose), constraints, and prohibitions.

Chapter 4 has presented the two Front-End modules implemented. Our tool includes two Front-End modules that manage access requests written in SQL and in SPARQL, respectively.

Chapter 5 has provided a description of the two Back-End modules implemented. Analogously to the Front-End, we have implemented two Back-End modules that support SQL and SPARQL, respectively.

Bibliography

[Alb] A. Albrecht. python-sqlparse. https://sqlparse.readthedocs.io/.

[Ber] T. Berners-Lee. The need for a universal syntax. https://www.w3.org/Addressing/URL/uri-spec.html.

[CWL+14] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J.J. Carroll, and B. McBride. RDF 1.1 concepts and abstract syntax. Technical report, W3C recommendation, 2014.

[DS20] S. De Capitani di Vimercati and P. Samarati. D3.3 – First Version of Policy Specification Language and Model. Technical report, MOSAICrOWN, June 2020.

[Gro12] OWL Working Group. Web Ontology Language (OWL), December 2012. https://www.w3.org/OWL/.

[HS13] S. Harris and A. Seaborne. SPARQL 1.1 query language, March 2013. https://www.w3.org/TR/sparql11-query.

[ISMR18a] R. Iannella, M. Steidl, S. Myles, and V. Rodriguez-Doncel. ODRL vocabulary & expression 2.2, February 2018. https://www.w3.org/TR/odrl-vocab/.

[ISMR18b] R. Iannella, M. Steidl, S. Myles, and V. Rodriguez-Doncel. ODRL vocabulary & expression 2.2, 2018. https://www.w3.org/TR/odrl-vocab/#encodings.

[IV18] R. Iannella and S. Villata. ODRL information model 2.2, February 2018. https://www.w3.org/TR/odrl-model.

[Tea20] The RDFLib Team. RDFLib: A Python library for working with RDF, 2020. https://github.com/RDFLib/rdflib.

[W3Ca] W3C JSON-LD Working Group. JSON for Linking Data. https://json-ld.org/.

[W3Cb] W3C Permissions and Obligations Expression Working Group. Requirements for an ODRL Processor. https://www.w3.org/2017/05/18-poe-minutes.

[W3Cc] W3C Wiki. LinkedData definition. https://www.w3.org/wiki/LinkedData.